CoScheme: Compositional Copatterns in Scheme

Paul Downen$^{1[0000-0003-0165-9387]}$ and
Adriano Corbelino II$^{1[0000-0002-6014-6189]}$

University of Massachusetts Lowell, Lowell MA 01854, USA
`Paul_Downen@uml.edu`
`Adriano_VilargaCorbelino@uml.edu`

Abstract. Since their introduction, copatterns have promised to extend functional languages — with their familiar pattern matching facilities — to synthesize and work with infinite objects through a finite set of observations. Thus far, their adoption in practice has been limited and primarily associated with specific tools like proof assistants. With that in mind, we aim to make copattern matching usable for ordinary functional programmers by implementing them as macros in the Scheme and Racket programming languages. Our approach focuses on composable copatterns, which can be combined in multiple directions and offer a new solution to the expression problem through novel forms of extensibility. To check the correctness of the implementation and to reason equationally about copattern-matching code, we describe an equational theory for copatterns with a sound, selective translation into λ-calculus.

Keywords: Codata · Copatterns · Scheme · Macros · Composition · Expression Problem.

1 Introduction

For decades, functional programmers have had a reliable and versatile method for representing tree-shaped structures: inductive data types. These can model data of any size — for example, lists of an arbitrary length — but each instance must be *finite*. Infinite information — like a stream of input that can go on forever — does not fit into an inductive type, so the programmer must use some other representation to model potentially infinite objects.

Fortunately, the inductive data types used by functional programmers every day have a polar opposite: *coinductive codata types*. The *coinductive* descriptor signals that values of the type may contain infinite information. Haskell programmers are already well-versed in coinductive styles of types since non-strict languages blur the line between induction and coinduction. For example, consider the usual example of the infinite list of Fibonacci numbers in Haskell:

```
fibs = 0 : 1 : zipWith (+) fib (tail fib)
```

`fibs` cannot be fully evaluated because it has no base case — it would eventually expand out to `0 : 1 : 1 : 2 : 3 : 5 : 8 : ...` forever — but this is no problem in a non-strict language that only evaluates as much as needed.

In contrast, *codata* describes types defined by primitive *destructors* that *use* values of the codata type, as opposed to the primitive constructors that define how to build values of a data type. For example, the usual `Stream a` codata type of infinite `a`'s is defined by two destructors: `Head : Stream a -> a` extracts the first element and `Tail : Stream a -> Stream a` discards the first element and returns the rest. To define new streams, we can describe how they react to different combinations of `Head` and `Tail` destructions using *copatterns* [2]. The copattern-based definition of the same `fibs` function above is:

```
fibs : Stream Nat
Head fibs = 0
Head (Tail fibs) = 1
Tail (Tail fibs) = zipWith _+_ fibs (Tail fibs)
```

Unfortunately, if we want to actually produce working code in this style, our choices are limited. Agda gives the most full-fledged implementation of copatterns in a real system [5]. However, Agda is primarily a proof assistant rather than a general-purpose programming language, and as such, has different concerns than an ordinary functional programmer. In particular, Agda currently does not understand if `fibs` is well-founded — it is — and so `fibs` is rejected by the termination checker. There is also some support for copatterns in OCaml [14], but as an unofficial extension that has not been merged into the main compiler.

We want to be able to write this kind of copattern-based code and have it fully integrated into a real, general-purpose programming language. The easiest way to do so is to start with a programmable programming language [9] — we focus in particular on Scheme and Racket — which offers a robust macro system for seamlessly implementing new language features. We show a new method for implementing copatterns as a collection of macros in order to program with copatterns in Scheme-like languages.

Rather than just recapping the previous macro-definition of copatterns [14], we focus on providing new methods of extensibility not available before. In particular, code defined by our copattern macros can be composed in a variety of ways, offering a new solution to the expression problem [19]. Not only do copatterns allow us to easily define code using equational reasoning in Scheme, but these new dimensions of composition also allow us to capture some "design patterns" used by functional programmers as first-class abstractions.

Our primary contributions are organized as follows:

– Section 2 shows examples of programming with copattern equations in Scheme-like languages, including new forms of program composition — vertical and horizontal — that allows us to solve familiar examples of the expression problem [19] through a fusion of functional and object-oriented techniques.
– Section 3 presents a theory for how to translate copatterns into a small core target language — untyped λ-calculus with recursion and patterns — with a local double-barrel transformation reminiscent of selective continuation-passing style transformation. Importantly, only the new language constructs are transformed, while existing ones in the target language are unchanged.

– Section 4 explains how to implement the high-level translation above in real code, and specifically how the implementation differs between Racket and a standard R^6RS-compliant Scheme.
– Section 5 demonstrates correctness in terms of an equational theory for reasoning about copattern-matching code in the source language, which is a conservative extension of the target language, and we prove that it is sound with respect to translation.

2 Programming with Composable Copatterns in Scheme

Let us consider some examples of programming by equational reasoning to get familiar with copatterns and how we can use them in Scheme. For example, even in a dynamically-typed language like Scheme, linked lists can be thought of as an inductively-defined type combining two constructed forms: `List a = null | (cons a (List a))`. Similarly, infinite streams can be understood as a procedure that exhibits two different behaviors at the same time: `Stream a = 'head -> a & 'tail -> Stream a`. In other words, any `Stream a` is a procedure that takes one argument, and its response depends on the exact value: given `'head` an `a` is returned, and given `'tail` another `Stream a` is returned.

In order to define new coinductive processes, one of the main entry points is the top-level, multi-line `define*` macro. This macro enables us to declare codata objects through a list of equations between a copattern on the left-hand side and an expression on the right-hand side. At the root of every copattern is a name for the object *itself*, which can be inside any number of applications — the applications may just list parameter names or more specific patterns narrowing down the concrete arguments that match. The simplest of a stream is all `zeroes` — whose `'head` is `0` and whose `'tail` is more `zeroes` — which can be defined as:

```
(define* [(zeroes 'head) = 0]
         [(zeroes 'tail) = zeroes])
```

Streams like `zeroes` are black boxes that can only be observed by passing `'head` or `'tail` as arguments to get their response. Still, this is enough for many useful operations, like taking the first `n` elements, which can be `define*`d as:

```
(define* [(takes s 0) = '()]
         [(takes s n) = (cons (s 'head) (takes (s 'tail) (- n 1)))])
```

A constant stream is not particularly useful; more interesting streams will change over time. For example, imagine a "stuttering" stream $(0, 0, 1, 1, 2, 2, 3, 3, \dots)$ that repeats numbers twice before moving on. This stream can be defined by copattern matching equations:

```
(define* [ ((stutter n) 'head)        = n]
         [(((stutter n) 'tail) 'head) = n]
         [(((stutter n) 'tail) 'tail) = (stutter (+ n 1))])
```

So that `(takes (stutter 1) 10) = '(1 1 2 2 3 3 4 4 5 5)`.[1] because the first and second elements — `((stutter n) 'head)` and `(((stutter n) 'tail) 'head)` respectively — return the same `n` before incrementing.

But why is `stutter` well-defined, and how can we understand its meaning? As in many functional languages, the `=` in code really implies equality between the two sides, and this equality still holds when we plug in real values for placeholder variables like `n`. So to determine the first `'head` element, of `(stutter 1)`, we match the left-hand side and replace it with the right to get `((stutter 1) 'head) = 1`. Similarly, the second element is `(((stutter 1) 'tail) 'head) = 1` as well. The third element is accessed by two `'tail` projections and then a `'head` as the nested applications `((((stutter 1) 'tail) 'tail) 'head)`, which doesn't exactly match any left-hand side. However, equality holds in any context, and the inner application `(((stutter 1) 'tail) 'tail)` *does* match the third equation. Thus, we can apply a few steps of equational reasoning to derive the expected answer `2`:

```
((((stutter 1) 'tail) 'tail) 'head) = ((stutter (+ 1 1)) 'tail)  ; line 3
                                    = ((stutter 2) 'head)        ; +
                                    = 2                          ; line 1
```

So these three examples work, but is every case really covered? The `Stream Nat` interface that `stutter`'s output follows allows for any number of `'tail` projections followed by a final application to `'head` that returns a natural number. `stutter` works its way through these projections in groups of two, eliminating a pair of `'tail` projections at a time until it gets to the end case, which is either a `'head` (if the total number of `'tail`s is even) or a `'tail` followed by `'head` (if the total number of `'tail`s is odd). So `stutter` behavior is defined no matter what is asked of it. Even with other observations like `takes`, which passes around partial applications of `stutter` as a first-class value, internally `stutter` only "sees" the `'head` and `'tail` applications from `takes`, and is dormant otherwise.

However, definitions by copatterns are useful for more programming tasks than just streams and other infinite objects. For example, consider the usual definition of a simple arithmetic expression evaluator in typed functional languages like Haskell and OCaml (we use Haskell syntax here):

```
data Expr = Num Int | Add Expr Expr

eval :: Expr -> Int
eval (Num n)   = n
eval (Add l r) = eval l + eval r
```

While Scheme does not have algebraic data types, we can encode them as a list starting with the constructor name as a quoted symbol and the arguments as the remainder of the list. So `Num 5` would be represented as the quoted list `'(num 5)`, and `Add l r` would be represented as the *quasiquote* `` `(add ,l ,r) `` which plugs in the values bound to variables `l` and `r` as the second and third elements of the

[1] Try it! The code provided in the supplemental materials implements `define*` and related macros. All examples shown here are executable Scheme and Racket code.

list (denoted by the "unquote" comma , before the variable names). We can then use the facilities of `define*` to write almost identical code in Scheme like so:

```
(define* [(eval `(num ,n))     = n]
         [(eval `(add ,l ,r)) = (+ (eval l) (eval r))])
```

Fantastic, it works! Both the Scheme and Haskell code have the same structure. And on the surface, they both share the same strengths and weaknesses. From the lens of the *expression problem* [19], it is easy to add new operations to existing expressions — such as listing the numeric literals in an expression

```
(define* [(list-nums `(num ,n))     = (list n)]
         [(list-nums `(add ,l ,r)) = (append (list-nums l) (list-nums r))])
```

— but adding new classes of expressions is hard. For example, if we wanted to support multiplication, we could add a `Mult` constructor to the `Expr` data type, but this would require modifying *all* existing operations and case-splitting expressions over `Expr` values. Even worse, if we wanted to support both expression languages — with or without multiplication — we would have to copy the code and maintain both versions.

Thankfully, our implementation of copattern matching in Scheme includes new facilities for composing code snippets compared to current functional (or object-oriented) languages. However, to avoid unwanted surprises, the programmer does have to ask for them. This is a small request, and can be done by replacing `define*` with `define-object`, such as:

```
(define-object
  [(list-nums* `(num ,n))     = (list n)]
  [(list-nums* `(add ,l ,r)) = (append (list-nums* l) (list-nums* r))])
```

The `list-nums*` object behaves exactly like `list-nums` in all the same contexts it works in, but in addition, it implicitly inherits additional functionality for composition defined elsewhere. This new composition lets us break existing multi-line definitions into individual parts, and recompose them later. For example, the evaluator can be composed in terms of separate objects for each line like so:

```
(define-object [(eval-num `(num ,n)) = n])
(define-object [(eval-add `(add ,l ,r)) = (+ (eval-add l) (eval-add r))])
(define eval* (eval-num 'compose eval-add))
```

So `(eval expr)` is the same as `(eval* expr)` for any well-formed expression argument. Why program in this way? Now, if we want to extend the functionality of existing operations — like evaluation and listing literals — to support a new class of expression, we can define the new special cases separately as a patch and then *compose* them with the existing code as-is like so:

```
(define-object [(eval-mul `(mul ,l ,r)) = (* (eval-mul l) (eval-mul r))])
(define-object [(list-mul `(mul ,l ,r)) = (append (list-mul l) (list-mul r))])

(define eval-arith (eval* 'compose eval-mul))
(define list-nums-arith (list-nums* 'compose list-mul))
```

So for an expression `(define expr1 '(add (mul (num 2) (num 3)) (num 4)))`, the extended code successfully yields the correct answers `(eval-arith expr1) = 10` and `(list-nums-arith expr1) = '(2 3 4)` whereas the original code fails at the `'mul` case.[2] Note that this composition automatically generates *new* functions and leaves the original code intact, which can still be used for the smaller expression language with only numbers and addition.

This example emphasizes our guiding principle: *composition*. We call combinations like `(eval-num 'compose eval-add eval-mul)` *vertical composition* since they behave as if we simply stacked their internal cases vertically, like in the original definition of `eval`.

Not all types of language extensions are this simple, though. Consider what happens if we want to support algebraic expressions which might have variables in them. To evaluate a variable, we need a given environment — mapping names to numbers — which we can use to look up the variable's value.

```
(define-object [(eval-var env `(var ,x)) = (lookup env x)])
```

However, it is wrong to just vertically compose this variable evaluator with the previous code because the arithmetic evaluator only takes a single expression as an argument, whereas the variable evaluator needs *both* an environment and an expression. The manual way to perform this extension is routine for functional programmers: in addition to adding a new case, we have to add an extra parameter to each case, which gets passed along on all recursive calls.

It would be highly disappointing to have to rewrite our existing code in-place to do this extension. Fortunately, our copattern language allows for another type of composition — *horizontal composition* — which allows us to combine sequences of steps, one after another, and automatically fall through to the next case if something fails. For this example, we can define a general procedure `with-environment` to perform the above transformation, taking any extensible evaluator object expecting just an expression and threading an environment along each recursive call. This lets us patch our existing arithmetic evaluator with an environment and then compose it with variable evaluation like so:

```
(define (with-environment eval-ext)
  (object [(self env expr)
          (with-self (override-lambda* self
                       [(_ sub-expr) = (self env sub-expr)])
            (try-apply-forget eval-ext expr))]))

(define eval-alg ((with-environment (eval-arith 'unplug)) 'compose eval-var))
```

The `with-environment` function is the most complex code we have seen so far, but it just spells out the usual steps a functional programmer uses to modify existing code with an environment.

[2] The astute reader might notice that for this to work, the recursive calls to `eval-mul` cannot be specifically tied to this definition because it only says what to do with multiplication and fails to handle the other cases. Instead, recursive calls to `eval-mul` must *also* open to invoking the other code associated with `eval-num` and `eval-add` even though it not known to be associated with them yet.

- Given the evaluator `eval-ext`, it returns a new first-class `object` (which is the same as `define-object` without assigning a name) that expects both an environment and expression to process.
- This new object then invokes `eval-ext` by passing just the expression, except that if `eval-ext` ever tries to recurse on itself with a sub-expression, the calls `(self sub-expr)` gets replaced with `(self env sub-expr)` just like the template transformation.
- This transformation of the evaluator's notion of self is done by the `with-self` operation, which can override the original recursive `self`.
- Finally, if none of the clauses of `eval-ext` succeed, then this updated evaluator also falls through as before, forgetting the application had ever happened via `try-apply-forget`.

The complete algebraic evaluator can then be made from an open-ended, extensible version of the arithmetic evaluator — retrieved from `(eval-arith 'unplug)` — horizontally composed to take an environment and vertically composed with the single-line `eval-var`. It can now successfully evaluate algebraic expressions, such as `(define expr2 '(add (var x) (mul (num 3) (var y))))`, so that running `(eval-arith '((x . 10) (y . 20)) expr)` returns `70` because the environment maps `x` to `10` and `y` to `20`.

Another possible way to evaluate expressions with variables is *constant folding*, a common optimization where operations are simplified unless they are blocked by variables whose values are unknown. In other words, the evaluator might return a blocked expression if it cannot fully calculate the final number. Ideally, we would like to extend our existing evaluator as-is, with the additional cases when blocked expressions are encountered. However, as written, the equation handling ``(eval `(add ,l ,r))`` already commits to a real numeric addition, even if evaluating `l` or `r` does not give a numeric result.

To avoid over-committing before we know whether evaluation will successfully calculate a final number or not, we can — for the first time — rewrite the basic clauses of evaluation in a more defensive style. Essentially, this splits evaluation into two separate steps: (1) check which operation we are supposed to do and evaluate the two sub-expressions, (2) combine the two expressions according to that operation. For example, the two steps for addition look like:

```scheme
(define-object eval-add-safe
  [(self 'eval ('add l r))
  = (self 'add (self 'eval l) (self 'eval r))]
  [(self 'add x y) (try-if (and (number? x) (number? y)))
  = (+ x y)])
```

Here, the evaluation step is explicated by a `'eval` tag, to help distinguish from the other operation `'add` for adding the left and right results. Note that in this code, the `'add` clause only performs a numeric addition `+` if it knows for sure that *both* of the arguments are actually numbers, check by the `try-if` guards. We can now compose the original base-case for evaluating numbers with this "safer" version of addition that fails to match cases where sub-expressions don't evaluate to numbers (multiplication could be added as well in a similar style):

```
(define eval-arith-safe (eval-num 'compose eval-add-safe))
```

So (eval-arith-safe expr1) still evaluates to 70, but (eval-arith-safe expr2) fails when it finds a variable sub-expression.

If it finds a variable, constant folding will just leave it alone and return an unevaluated expression rather than a final number. Because the 'eval operation might return a (partially) unevaluated expression, we now need to handle cases where the left or right (or both) sub-expressions do not evaluate to numbers. In each of those cases, we must reform the addition expression out of what we find, converting numbers n into a syntax tree of the form `(num ,n).

```
(define-object [(leave-variables 'eval ('var x)) = (list 'var x)])

(define-object reform-addition
  [(reform 'add l r) (try-if (number? l)) = (reform 'add `(num ,l) r)]
  [(reform 'add l r) (try-if (number? r)) = (reform 'add l `(num ,r))]
  [(reform 'add l r)                      = (list 'add l r)])
```

The final constant-folding algorithm can be composed from this "safe" version of evaluation, along with the cases for leaving variables alone and reforming partially-evaluated additions:

```
(define constant-fold
  (eval-arith-safe 'compose leave-variables reform-addition))
```

So now (constant-fold 'eval expr2) successfully returns expr2 itself (because there are no operations to perform without knowing the values of variables x and y). And running (constant-fold 'eval expr3) on the expression

```
(define expr3 '(add (add (num 1) (num 1))
                    (mul (var x)
                         (mul (num 2) (add (num 2) (num 3))))))
```

simplifies it down to '(add (num 2) (mul (var x) (num 10))). To add other operations, like multiplication, we can easily define similar eval-mul-safe and reform-multiplication, and 'compose them with constant-fold without having to rewrite any code. All examples shown here are in the supplemental materials.

3 Translating Composable Copatterns

3.1 Challenges

Even though the behavior of small examples may be straightforward to understand, there are several challenges to correctly implementing copatterns in the general case. Some of these challenges are specific to Scheme — a dynamically-typed, call-by-value language — which forces us to carefully resolve the timing of when and which copatterns are matched. Other challenges are specific to our extensions to copatterns — the ability to compose copattern matching in two different directions — which also brings in the notion of the recursive "self."

Timing and the order of copattern matching Because we cannot gain any information from a static type system, we must interpret the programmer's code as it is written. This means we have to deal with copatterns that may have ambiguous cases where two different overlapping copattern equations match the same application. For example, this function moves a number by `1` away from `0` — positives are incremented and negatives are decremented:

```
(define* [(away-from0 x) (try-if (>= x 0)) = (+ x 1)]
         [(away-from0 x) (try-if (<= x 0)) = (- x 1)])
```

The two different equations overlap for `0` itself: either one matches the call `(away-from0 0)`. To disambiguate overlapping copatterns, the listed equations are always tried top-down, and the first full match "wins," as is typical in functional languages. In this case, the first line wins, so `(away-from0 0)` is `1`. Furthermore, guards like `try-if` and `try-match` are run left-to-right with shortcircuiting — the moment a copattern or a guard fails, everything to the right is skipped. This makes it possible to protect potentially-erroneous guards with another safety guard to its left, such as `(try-if (not (= y 0)))` followed by `(try-if (> (/ x y) z))`.

However, there are more timing issues besides these usual choices for disambiguation and short-circuiting. First of all, since we are in a call-by-value language, we have to handle cases where an object is used in a context that doesn't fully match a copattern *yet*, but could in the future — and possibly multiple different times. This can happen for instances like curried functions that take arguments in multiple different calls. Just like with ordinary curried functions, using such an object in a calling context passing only the first list of arguments — but not the second — builds a *value* which closes over the parameters so far. For example, consider this simple counter object that can add or get its current internal state.

```
(define* [((counter x) 'add y) = (counter (+ x y))]
         [((counter x) 'get)   = x])
```

The call `((counter 4) 'get)` matches the second equation, which is `4`, but `(counter 4)` on its own is not enough information to definitively match either copattern, so it is just a value remembering that `x = 4` and waiting for another call. Similarly, the call `(counter (+ x y))` on the right-hand side is *also* incomplete in the same sense, so it, too, is a value. This definition gives us an object with the following behavior:

```
> (define c0 (counter 4))
> (define c1 (c0 'add 1))
> ((c1 'add 2) 'get)
7
> (c1 'get)
5
```

So far, what we have seen so far seems similar to pattern-matching functions in languages that are curried-by-default. One way in which copatterns generalize curried functions is that each equation can take a *different* number of arguments. For example, consider this reordering of the `stutter` stream from section 2:

```
(define* [(((stutter n) 'tail) 'tail) = (stutter (+ n 1))]
         [(((stutter n) 'tail) 'head) = n]
         [ ((stutter n) 'head)        = n])
```

Since none of the copatterns overlap, its behavior is exactly the same as before. But notice the extra complication here: calling `((stutter 10) 'head)` with two arguments (`10` and `'head`) should immediately return `10`. However, the first equation is waiting for three arguments (an `n` and two `'tail`s passed separately). That means the underlying code implementing `stutter` *cannot* ask for three arguments in three different calls and then check that the last two are `'tail`. Instead, it has to eagerly match the arguments its given against the patterns and try each of the guards to see if the current line fails — and only *after* that all succeed, it may ask for more arguments and continue the copattern match.

Composition and the dimensions of extensibility The second set of challenges is due to the new notions of object composition that we develop here. In particular, we want to be able to combine objects in two different directions:

– *vertical composition* is an "either or" combination of two or more objects, such as `(o1 'compose o2 ...)` that acts like `o1` or `o2`, *etc,* depending on which one knows how to respond to the context. Textually, vertical composition of `(object line-a1 ...)` and `(object line-b1 ...)` behaves as if we copied all each line of copattern-matching equations internally used to define the two objects and pasted them vertically into the newly-composed object as:
```
(object line-a1 ...
        line-b1 ...)
```
– *horizontal composition* is an "and then" combination of objects in a copattern-matching line, such as `[(self 'method1) (try-object o1)]` defining a `'method1` that continues to act like `o1` when `o1` knows how to respond to the surrounding context, and otherwise tries the next line. Textually, the vertical composition of a `'method1` followed by trying another object with its own copattern-matching contexts `Q1 Q2 ...` acts as if the two copatterns are combined, and the inner object is inlined into the outer one like so:
```
(object [(self 'method1) (try-object (object [Q1 = response1]
                                             [Q2 = response2]
                                             ...))]
        ...)
=
(object [(self 'method1) (comatch Q1) = response1]
        [(self 'method1) (comatch Q2) = response2]
        ...)
```

Even though we can visually understand the two directions of composition by the textual manipulations above, in reality, both of these compositions are done at run-time (*i.e.,* with arbitrary procedural values), as opposed to "compile-time" transformation (*i.e.,* macro-expansion time manipulations of code). This means we need an extensible representation of run-time object values that allows for automatically switching from one object to another in the case of copattern-match failure, as well as correctly keeping track of what to try next.

The basic idea of this representation can be understood as an extension of an idiom in ordinary functional programming. In order to define an open-ended, pattern-matching function, we can give the cases we know how to handle now by matching on the arguments and include a default "catch-all" case at the end for the other behavior. In Haskell, this might look like

```
f next PatA1 PatA2 ... = expr1
f next PatB1 PatB2 ... = expr2
...
f next x1    x2    ... = next x1 x2 ...
```

For example, consider the single-line `eval-add` evaluator object from section 2. In order to compose `eval-add` with another evaluator handling a different case, like `eval-add`, its internal extensible code takes an extra hidden argument saying what to try `next` if its line does not match, analogous to:

```
(define (eval-add-ext1 next)
  (lambda* [(self 'eval `(add ,l ,r)) = (+ (self 'eval l) (self 'eval r))]
           [ self                     = (next self)]))
```

Note that, unlike the Haskell code above, the hidden `next` parameter also takes *another* hidden parameter: `self`. Why? Because if the `next` set of equations needs to recurse, it cannot actually jump to itself directly — that would skip the `eval-add` code entirely — but needs to jump back to the very first equation to try. This `self` parameter holds the value of the *whole* object after all compositions have been done, as it appeared in the original call site. Thus, the internal extensible code `eval-add-ext` *also* takes this second `self` parameter for the same reason: it may be the second component of a composition, and can be further expanded into built-in Racket primitives like so:

```
(define ((eval-add-ext2 next) self)
  (match-lambda* [(list 'eval `(add ,l ,r)) (+ (self 'eval l) (self 'eval r))]
                 [args                      ((next self) args)]))
```

3.2 Double-barrel translation

To explain the correctness and behavior of composable copattern matching, we give a high-level translation into a conventional λ-calculus with recursion and pattern matching (given in fig. 1). Our pattern language is modeled after a small common core found among various implementations of Scheme, which includes normal variable wildcards x that can match anything, quoted symbols **'**x, and lists of the form null or $(\mathrm{cons}\, P\, P')$. Note that we assume all bound variables x in a pattern are distinct. As shorthand, we write a list of patterns

$$\begin{aligned} \mathit{Term} \ni M, N &::= x \mid M\ N \mid \lambda x.M \mid K \mid \mathbf{match}\, M\, \mathbf{with}\, \{\, P \to N ... \,\} \mid \mathbf{rec}\, x = M \\ \mathit{Pattern} \ni \quad P &::= x \mid \texttt{'}x \mid \mathrm{null} \mid \mathrm{cons}\, P\, P' \end{aligned}$$

Fig. 1. Target language: pure λ-calculus with pattern-matching and recursion.

$P_1\ P_2\ \ldots\ P_n$ for $(\mathrm{cons}\, P_1\ (\mathrm{cons}\, P_2\ \ldots (\mathrm{cons}\, P_n\, \mathrm{null})))$. To model the patterns found in typed functional languages like ML and Haskell, such as constructor applications $K\ P...$, we can represent the constructor as a quoted symbol $\texttt{'}K$ and the application as a list $\texttt{'}K\ P...$. The patterns' specifics are surprisingly not essential to the main copattern translation and could be extended with other features found in more specific implementations.

For simplicity, this translation begins from a small source language with copatterns (given in fig. 2) separated into three main syntactic categories:

(M, N) *Terms* represent ordinary first-class values as well as applications. The new forms of terms are $\lambda^* B$, which gives a self-referential copattern-matching object, along with $\mathbf{template}\, B$ and $\mathbf{extension}\, O$ which include the other two syntactic categories as first-class values.

(B) *Templates* represent self-referential code without a fixed self. Instead, the "self" placeholder remains unbound for now, and it can be instantiated later as $\mathbf{template}\, B\ V$ (where the "self" of the template is bound to V) or $\lambda^* B$ (where the "self" of the template is recursively bound to $\lambda^* B$).

(O) *Extensions* represent extensible code that can be composed together both vertically and horizontally. Instead of failing on an unsuccessful match, will try an as-of-yet unspecified "next" option. To support recursion, the "self" placeholder is also unbound for now — just like with templates — and can be bound later when the whole object is finished being composed. The "next" thing to try can be given by the vertical composition with another extension $O; O'$ or a base-case template $O; B$. Arbitrary first-class values can passed in as the next option (V) and the self object (W) as $\mathbf{extension}\, O\ V\ W$.

The remaining new syntax gives ways to define and combine copattern-matching expressions. Copatterns $Q[x]$ themselves are a subset of contexts, Q, surrounding an object internally named x. Two lines separated by a semicolon $(O; O')$ is vertical composition that tries either O or O', and prefixing with a copattern-matching expression $(Q[x]O)$ is horizontal composition that tries $Q[x]$ and then O. The ε represents an empty extension with respect to vertical composition: it immediately refers to the next option. Smaller special cases of matching include pattern lambdas $(\lambda P.O)$ that try to match a new argument against P, and pattern guards $(\mathbf{match}\, P \leftarrow M\ O)$ that try to match a given expression M against P; both of which continue as O if they succeed.

Finally, we have the terminators for ending a sequence of matching. A template can end in the empty ε (which just fails, because there is no code to handle the case) or a $\mathbf{continue}\, x \to M$ which serves as the default "catch-all" case. The

$$\begin{aligned}
\mathit{Term} \ni M, N &::= \cdots \mid \lambda^{*} B \mid \mathbf{template}\, B \mid \mathbf{extension}\, O \\
\mathit{Template} \ni \quad B &::= \varepsilon \mid O; B \mid \mathbf{continue}\, x \to M \\
\mathit{Extension} \ni \quad O &::= \varepsilon \mid O; O' \mid Q[x]\ O \mid \lambda P.\ O \mid \mathbf{match}\, P \leftarrow M\ O \mid \mathbf{try}\, x \to B \\
\mathit{Copattern} \ni \quad Q &::= \square \mid Q\ P \\
\mathit{Pattern} \ni \quad P &::= x \mid \texttt{'}x \mid \mathrm{null} \mid \mathrm{cons}\, P\, P'
\end{aligned}$$

Syntactic sugar:

$$\begin{aligned}
\mathbf{else}\, M &= \mathbf{continue}\ \_ \to M & (= M) &= \mathbf{do}\, M = \mathbf{try}\ \_ \to \mathbf{else}\, M \\
\mathbf{if}\, M\ O &= \mathbf{match}\, \mathrm{True} \leftarrow M\ O & (\mathbf{let}\, x = M\ O) &= \mathbf{match}\, x \leftarrow M\ O
\end{aligned}$$

Fig. 2. Source language: target extended with nested copatterns, self-referential objects, recursion templates, and composable extensions.

parameter x bound by $\mathbf{continue}\, x \to M$ is another way to introduce a name for the recursive reference to the object itself at the end of a template and allows for M to restart from the top and continue the computation. The familiar syntactic sugar $\mathbf{else}\, M$ covers the common case where M give an answer without recursively continuing. Similarly, an extension can end with $\mathbf{try}\, x \to B$. This gives a "catch-all" case that runs some other (non-extensible) template B. The parameter x bound by $\mathbf{try}\, x \to B$ gives a name to the next option that would have been tried after this one and allows B to explicitly move on to the next option if it needs to. The syntactic sugar $\mathbf{do}\, M$ covers the most common case of **try** which definitively commits to a particular term M to return as the result without trying any further options. To write examples in a similar style to ML-family languages, we also use the syntactic sugar $(= M)$ with the same meaning, which looks odd out of context but expresses the equational nature of copattern matching when used in examples.

Thus, the full translation from the source (fig. 2) to target (fig. 1) is given in fig. 3. This translation shares many similarities to continuation-passing style (CPS) translations. However, we explicitly avoid converting the entire program to CPS. Notably, every syntactic form for the source language is unchanged; for example, $[\![M\ N]\!] = [\![M]\!]\ [\![N]\!]$. Instead, the only time we need to introduce an extra parameter is for the two new syntactic categories. All templates are translated to functions that take a value for the whole object itself to a new version of that object. Similarly, all extensions are translated to functions that take both a template as the "base case" to try next and a value for the whole object itself. Even though this is dynamically-typed, we can view the type of templates as object transformers and extensions as template transformers:

$$\begin{aligned}
\mathit{Object} &= \text{some type of function} \\
\mathit{Template} &= \mathit{Object} \to \mathit{Object}' \\
\mathit{Extension} &= \mathit{Template} \to \mathit{Template}' = \mathit{Template} \to \mathit{Object} \to \mathit{Object}'
\end{aligned}$$

Translating new terms:

$$\begin{aligned}
[\![\lambda^* B]\!] &= (\mathbf{rec}\, \mathit{self} = T[\![B]\!]\ (\lambda x.\mathit{self}\ x)) && =_\eta (\mathbf{rec}\, \mathit{self} = T[\![B]\!]\ \mathit{self}) \\
[\![\mathbf{template}\, B]\!] &= T[\![B]\!] \\
[\![\mathbf{extension}\, O]\!] &= E[\![O]\!] \\
[\![M]\!] &= \text{by induction} && \text{(otherwise)}
\end{aligned}$$

Translating templates:

$$\begin{aligned}
T[\![\varepsilon]\!] &= \lambda s.\mathit{fail}\ s && =_\eta \mathit{fail} \\
T[\![O; B]\!] &= \lambda s.E[\![O]\!]\ T[\![B]\!]\ s && =_\eta E[\![O]\!]\ T[\![B]\!] \\
T[\![\mathbf{continue}\, x \to M]\!] &= \lambda x.[\![M]\!]
\end{aligned}$$

Translating copattern-matching and pattern-matching functions:

$$\begin{aligned}
E[\![(Q[x]\ P)\ O]\!] &= E[\![Q[x]\ (\lambda P.O)]\!] \\
E[\![x\ O]\!] &= \lambda b.\lambda x.E[\![O]\!]\ b\ x \\
E[\![\lambda P.O]\!] &= E[\![\lambda x.\, \mathbf{match}\, P \leftarrow x\ O]\!] && (\text{if } P \notin \mathit{Variable})
\end{aligned}$$

Translating other extensions:

$$\begin{aligned}
E[\![\varepsilon]\!] &= \lambda b.\lambda s.b\ s && =_\eta \lambda b.b \\
E[\![O; O']\!] &= \lambda b.\lambda s.E[\![O]\!]\ (E[\![O']\!]\ b)\ s && =_\eta E[\![O]\!] \circ E[\![O']\!] \\
E[\![\lambda x.O]\!] &= \lambda b.\lambda s.(\lambda x.E[\![O]\!]\ (\lambda s'.b\ s'\ x)\ s) \\
E[\![\mathbf{match}\, P \leftarrow M\ O]\!] &= \lambda b.\lambda s.\, \mathbf{match}\, [\![M]\!]\, \mathbf{with}\, \{\, P \to E[\![O]\!]\ b\ s;\ \_ \to b\ s\,\} \\
E[\![\mathbf{try}\, x \to B]\!] &= \lambda x.T[\![B]\!]
\end{aligned}$$

Fig. 3. Translating copattern-based source code to the target language.

The interesting cases for translating terms are the new forms. **template** B and **extension** O are just translated to their given forms as transformation functions. With $\lambda^* B$, we need to recursively plug its translation in for its self parameter. Note the one detail that the recursive *self* is η-expanded to in this application. This ensures that $\lambda x.\mathit{self}\ x$ is treated as a value in a real implementation, and is always safe assuming that B describes a function (non-functional cases of $\lambda^* B$ are undefined user error).

For templates and extensions, the terminators **continue** and **try** are translated to plain λ-abstractions that allow the programmer direct access to their implicit parameters. Complex copatterns ($x\ P_1 ... P_n\ O$) are reduced down to a simpler sequence of pattern lambdas ($x\ \lambda P_1. \dots \lambda P_n.\ O$), and pattern lambdas ($\lambda P.O$) are reduced down to a simpler non-matching lambda followed by an explicit match guard ($\lambda x.\, \mathbf{match}\, P \leftarrow x\ O$).

This leaves just the base cases of simple extension forms. Each time an extension (of form $\lambda b.\lambda s. \dots$) "fails," it calls the given next template with the given self object ($b\ s$). A match guard $[\![\mathbf{match}\, P \leftarrow M\ O]\!]$ will try to match the translation of M against the pattern P; the success case continues as $E[\![O]\!]$ with the same next template and self. A non-matching lambda $[\![\lambda x.O]\!]$ always succeeds (for now), but note that the next template to try on failure has to be changed to include the given argument. Why? Because the lambda has already consumed the next argument from its context, it would be gone if, later on, the following operations fail and move on to the next option. So instead of invoking the given b directly as $b\ s'$ (for a potentially different future s'), they need to invoke b applied to this argument x as $b\ s'\ x$.

In this translation, we also give the η-reduced forms on the right-hand side when available. This shows that the empty extension ε is just the identity function (given the next thing b to try, ε does nothing and immediately moves on to b), and horizontal composition $O; O'$ is just ordinary function composition.

4 Macro Definition

The real implementation of copattern matching in the Scheme macro system is quite similar to the high-level translation given in fig. 3 However, there are some important differences which have to do with integrating the new feature with the rest of the language, as well as practical implementation details. For example, note the definition of $[\![\lambda^* B]\!]$ in particular. While the η-equality simplifying $\lambda x.\mathit{self}\ x$ to just self is theoretically sound, it does not work in practice: when a Scheme interpreter tries to evaluate the right-hand side $(T[\![B]\!]\ \mathit{self})$ of the recursive binding, it first tries to lookup the value bound to self which has not been defined yet, leading to an error. This one level of η-expansion delays the evaluation step so that $\lambda x.\mathit{self}\ x$ returns a closure around the location where self will be placed, which is passed to $T[\![B]\!]$ whose result is bound to self.

Happily, instead of a single big recursive macro, first-class templates and extensions make it possible to implement the various parts of copattern matching as many independent macros that can be used separately and composed by the programmer. For example, $\lambda P.O$, **if** $M\ O$, **match** $P \leftarrow M\ O$, *etc.* are all implemented as self-contained macros that create new extension values around other extensions. These forms need to be macros because they either bind variables around an expression (like λP or **match**) or do not evaluate a sub-expression in some cases (like **if**). Other simpler forms, like the empty object or the composition $O; O'$, are just ordinary procedural values and not defined as macros. The macro for copattern matching, $Q[x]\ O$, is the only main recursive step, which decomposes a copattern into a sequence of more basic matching λs.

Additionally, the source language, as implemented, is more flexible than presented in fig. 2, in the sense that there are not as many syntactic categories. So the O in forms like $\lambda P.O$ or **if** $M\ O$ can be *any* host language expression as long as it evaluates to a procedure following the calling convention of extensions (otherwise a run-time error may be encountered). The implementation also supports other standard Scheme expressions, including functions of multiple arguments (corresponding to `(lambda (P ...) O)` or the copattern `(self P ...)`) and variable numbers of arguments (corresponding to `(lambda (P ... . rest) O)` or the copatterns `(self P ... . rest)` or `(apply self P ... rest)`). The main points where the syntactic restrictions are used are in the macros implementing **extension** O or **template** B. For example, the `extension` macro definition is:

```
(define-syntax-rule
  (extension [copat step ...] ...)
  (merge [chain (comatch copat) step ...] ...))
```

where `merge` is the regular definition of first-class function composition, `comatch` is the macro for the copattern matching form $Q[self]\ O$, and `chain` is a macro for right-associating any chain of operations to avoid overly-nested parentheses, with special support for unparenthesized terminators:

```
(define-syntax chain
  (syntax-rules (= try)
    [(chain ext)                   ext]
    [(chain (op ...) step ... ext) (op ... (chain step ... ext))]
    [(chain = expr)                (always-do expr)]
    [(chain try ext)               ext]))
```

One concern for a real implementation is to consider what kind of pattern-matching facilities the host language already provides. Unfortunately, the answer is not standard across different languages in the Scheme family. For example, the R^{6}RS standard does not require any built-in support for pattern matching to be fully compliant, but specific languages like Racket may include a library for pattern matching by default. Thus, we provide two different implementations to illustrate how copatterns may be implemented depending on their host language:

– A Racket implementation that uses its standard pattern-matching constructs `match` and `match-lambda*`. Thus, the **match** from the target language in fig. 1 is interpreted as Racket's `match`, and the translation of $E[\![\lambda P.O]\!]$ is implemented directly as `match-lambda*` instead of separating the λ from the pattern as in fig. 3. This choice ensures the pattern language implemented is exactly the same as the pattern language already used in Racket programs.
– A general implementation intended for any R^{6}RS-compliant Scheme,[3] which internally implements its own pattern-matching macro, `try-match`, by expanding into other primitives like `if` and comparison predicates. Of note, due to only having to handle a single line of pattern-matching at a time, this implementation is 75 lines of Scheme and supports quasiquoting forms of patterns. This gives a sufficiently expressive intersection between Racket's pattern-matching syntax and the manually implemented R^{6}RS version.

5 Correctness

We already used the translation to a core λ-calculus as a specification for implementing compositional copatterns, but the translation is also useful for another

[3] We have explicitly tested this implementation against Chez Scheme.

$$
\begin{array}{rl}
\mathit{Value} \ni V, W ::= & x \mid \lambda x.M \mid \mathrm{null} \mid \mathrm{cons}\, V\, W \mid \texttt{'}x \\
\mathit{EvalCxt} \ni \quad E ::= & \Box \mid E\ M \mid V\ E \mid \mathbf{match}\, E\, \mathbf{with}\, \{\, P \to N ... \,\} \mid \mathbf{rec}\, x = E
\end{array}
$$

$$
\begin{array}{llll}
(\beta) & (\lambda x.M)\ V = M[V/x] & & \\
(\mathit{match}) & \mathbf{match}\, V\, \mathbf{with}\, \{\, P \to N;\ P' \to N' ... \,\} & = N[W.../x...] & (\text{if } P[W.../x...] = V) \\
(\mathit{apart}) & \mathbf{match}\, V\, \mathbf{with}\, \{\, P \to N;\ P' \to N' ... \,\} & = \mathbf{match}\, V\, \mathbf{with}\ \{\, P' \to N' ... \,\} & (\text{if } P \mathrel{\#} V) \\
(\mathit{rec}) & (\mathbf{rec}\, x = V) = V[(\mathbf{rec}\, x = V)/x] & &
\end{array}
$$

Apartness between patterns and values ($P \mathrel{\#} V$):

$$
\frac{V \notin \mathit{Variable} \cup \{\, \texttt{'}x \,\}}{\texttt{'}x \mathrel{\#} V}
\qquad
\frac{V \notin \mathit{Variable} \cup \{\, \mathrm{null} \,\}}{\mathrm{null} \mathrel{\#} V}
$$

$$
\frac{V \notin \mathit{Variable} \cup \{\, \mathrm{cons}\, W\ W' \mid W, W' \in \mathit{Value} \,\}}{\mathrm{cons}\, P\ P' \mathrel{\#} V}
$$

$$
\frac{P \mathrel{\#} W}{\mathrm{cons}\, P\ P' \mathrel{\#} \mathrm{cons}\, W\ W'}
\qquad
\frac{P' \mathrel{\#} W'}{\mathrm{cons}\, P\ P' \mathrel{\#} \mathrm{cons}\, W\ W'}
$$

Fig. 4. Untyped equational axioms of the target language.

purpose: checking the expected meaning of copattern-matching code. With that in mind, we now look for some laws that let us equationally reason about some programs to make sure they behave as expected.

First, the core target language — a standard call-by-value λ-calculus extended with pattern-matching and recursion — has the equational theory shown in fig. 4, which is the *reflexive*, *symmetric*, *transitive*, and *compatible* (*i.e.,* equalities can be applied in *any* context) closure of the listed rules. It has the usual β axiom (restricted to substituting value arguments), two axioms for handling pattern-match success (*match*) and failure (*apart*), and an axiom for unrolling recursive values (*rec*). Values (V, W) include the usual ones for call-by-value λ-calculus (x and $\lambda x.M$) as well as lists (null and cons V W) and symbolic literals ($\texttt{'}x$). Matching a value V against a pattern P will succeed if the variables (x...) in the pattern can be replaced by other values (W...) to generate exactly that V: $P[W.../x...] = V$. In contrast, matching fails if the two are known to be *apart*, written $P \mathrel{\#} V$ and defined in fig. 4, which implies that all possible substitutions of P will *never* generate V. Note that while matching and apartness are mutually exclusive, there are some values that are neither matching nor apart from some patterns. For example, compare the value x against the pattern null; x may indeed stand for null or another value like $\lambda y.M$.

$$\begin{aligned}
\mathit{ExtensionFunc} \ni F &::= Q[x\ P]\ O \mid \lambda P.O \\
\mathit{Value} \ni V &::= \cdots \mid \lambda^*(F; B) \mid \textbf{template}\, B \mid \textbf{extension}\, O
\end{aligned}$$

Identity, associativity, and annihilation laws of composition:

$$\begin{array}{ccc}
\varepsilon; O = O & (O_1; O_2); O_3 = O_1; (O_2; O_3) & \textbf{do}\, M; O = \textbf{do}\, M \\
\varepsilon; B = B & (O_1; O_2); B = O_1; (O_2; B) & \textbf{do}\, M; B = \textbf{else}\, M
\end{array}$$

Pattern and copattern matching:

$$\begin{aligned}
\textbf{match}\, P \leftarrow V\ O &= O[W.../x...] && (\text{if } P[W.../x...] = V) \\
\textbf{match}\, P \leftarrow V\ O &= \varepsilon && (\text{if } P \# V) \\
(\textbf{template}\, (\lambda P.\, \textbf{do}\, M); B)\ V'\ V &= M[W.../x...] && (\text{if } P[W.../x...] = V) \\
(\textbf{template}\, (\lambda P.O); B)\ V'\ V &= (\textbf{template}\, B)\ V'\ V && (\text{if } P \# V) \\
C[(\textbf{template}\, (Q[y] = M); B)\ V] &= M[V/y][W.../x...] && (\text{if } Q[W.../x...] = C) \\
C[(\textbf{template}\, (Q[y]\ O); B)\ V] &= C[(\textbf{template}\, B)\ V] && (\text{if } Q \# C) \\
C[\lambda^*(Q[y] = M); B] &= M[(\lambda^*(Q[y] = M); B)/y] && (\text{if } Q[W.../x...] = C) \\
&\quad [W.../x...] \\
C[\lambda^*(Q[y]\ O); \textbf{else}\, M] &= C[M] && (\text{if } Q \# C)
\end{aligned}$$

Apartness between copatterns and contexts ($Q \# C$):

$$\frac{Q[W.../x...] = C \quad P \# V}{Q\ P \# C\ V} \qquad \frac{Q \# C}{Q\ P \# C} \qquad \frac{Q \# C}{Q \# C\ V}$$

Fig. 5. Some equalities of copattern extensions.

The first usual property is that the translation specified in fig. 3 is a *conservative extension*: any two terms that are equal by the target equational theory are still equal after translation. Because the translation is hygienic and compositional by definition, we can follow the proof strategy in [8].

Proposition 1 (Conservative Extension). *If $M = N$ in the equational theory of the target (fig. 4), then so too does $[\![M]\!] = [\![N]\!]$.*

To reason about the new features in the source language — introduced by λ^*, **template**, and **extension** — we introduce additional axioms given in fig. 5, so that the source equational theory is the *reflexive*, *symmetric*, *transitive*, and *compatible* closure of these rules in both figs. 4 and 5. The purpose of these new equalities is to perform some reasoning about programs using copatterns, and in particular, to check that the syntactic use of = really means equality. For example, a special case is $Q[\lambda^*(Q[y] = M); B] = M[\lambda^*(Q[y] = M); B/y]$, which says a λ^* appearing in *exactly* the same context as the left-hand side of an equation will unroll (recursively) to the right-hand side. Other equations describe algebraic laws of copattern alternatives and how to fill in templates and extensions when applied. This source equational theory is *sound* with respect to translation.

Proposition 2 (Soundness). *The translation is* sound *w.r.t. the source and target equational theories (*e.g., *$M = N$ in fig. 5 implies $[\![M]\!] = [\![N]\!]$ in fig. 4).*

6 Related Work and Conclusion

Previously, copatterns have been developed exclusively from the perspective of statically-typed languages. Much of the work has been for dependently typed languages like Agda [5], which use a type-driven approach to elaborate copatterns [16,17]. The closest related work is the implementation of copatterns as an OCaml macro [14], but this, too, is concerned with type system ramifications. Here, we show how to implement copatterns with no typing information and focus instead on composition and equational reasoning.

The translation in fig. 3 is reminiscent of "double-barrelled CPS" [18] used to define control effects like delimited control [7] and exceptions [13]. In our case, rather than a "successful return path" continuation, there is a "resume recursion" continuation. Expressions that return successfully just return as normal, similar to a selective CPS [15], which makes the it possible to implement as a macro expansion. A "next case" continuation — to handle copattern-matching failure — is introduced to make each line of a copattern-based definition a separate first-class value. From that point, the "recursive self" must be a parameter because no one sliver of a definition suffices to describe the whole.

Theories of object-oriented languages [1,6] also model the "self" keyword as a parameter later instantiated by recursion; either as an explicit recursive binding, or encoded as self-application. This is done to handle the implicit composition of code from inheritance, whereas here, we need to handle explicit composition of first-class extensible objects. The full connection between copatterns — as we describe here — and object-oriented languages remains to be seen. In terms of the lisp family of languages, the approach here seems closest to a first-class generalization of *mixins* [3,10] with a simple dispatch mechanism (matching), in contrast to class-based frameworks focused on complex dispatch [11,12,4].

We have shown here how to implement a more extensible, compositional version of copatterns as a macro in standard Scheme as well as Racket. Our major focus involves new ways to compose (co)pattern matching code in multiple directions — vertically and horizontally — which can be used to solve the expression problem since it can encode certain functional and object-oriented design patterns. Despite the more general forms of program composition, we still support straightforward equational reasoning to understand code behavior, even when that code is assembled from multiple parts of the program. This equational reasoning is formalized in terms of an extended λ-calculus, which is soundly translated into a common core calculus familiar to functional programmers; we leave the equation of a *complete* and minimal equational theory for copatterns as future work.

Our work here does not include static types, inherited from Scheme's nature as a dynamically typed language. As future work, we intend to develop a type system for the copattern language described here; specific challenges include correctly specifying type types of (de)composed code as well as coverage analysis that ensures every case is handled after the composition is finished. The second direction of future work is to incorporate effects into copattern definitions and their equational reasoning, for example, subsuming (delimited) control operators into the copattern language as a way of expressing compositional effect handlers.

Acknowledgments. This material is based upon work supported by the National Science Foundation under Grant No. 2245516.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Springer (1996)
2. Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: Copatterns: Programming infinite structures by observations. In: POPL (2013)
3. B., G., C., W.R.: Mixin-based inheritance. In: OOPSLA/ECOOP (1990)
4. Chambers, C.: Object-oriented multi-methods in Cecil. In: ECOOP (1992)
5. Cockx, J., Abel, A.: Elaborating dependent (co)pattern matching. Proceedings of the ACM on Programming Languages **2**(ICFP) (2018)
6. Cook, W.R., Palsberg, J.: A denotational semantics of inheritance and its correctness. Inf. Comput. **114**(2), 329–350 (1994)
7. Danvy, O., Filinski, A.: Abstracting control. In: LFP (1990)
8. Downen, P., Ariola, Z.M.: Compositional semantics for composable continuations: From abortive to delimited control. In: ICFP (2014)
9. Felleisen, M., et al.: A programmable programming language. Commun. ACM **61**(3) (2018)
10. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and mixins. In: POPL (1998)
11. Gabriel, R., White, J., Bobrow, D.: CLOS: Integrating object-oriented and functional programming. Commun. ACM **34** (1991)
12. Ingalls, D.H.H.: A simple technique for handling multiple polymorphism. In: OOPSLA (1986)
13. Kim, J., Yi, K., Danvy, O.: Assessing the overhead of ML exceptions by selective CPS transformation. BRICS Report Series **5** (1998)
14. Laforgue, P., Régis-Gianas, Y.: Copattern matching and first-class observations in OCaml, with a macro. In: PPDP (2017)
15. Nielsen, L.R.: A selective CPS transformation. In: MFPS. vol. 45 (2001)
16. Setzer, A., Abel, A., Pientka, B., Thibodeau, D.: Unnesting of copatterns. In: RTA-TLCA (2014)
17. Thibodeau, D.: Programming Infinite Structures using Copatterns. Master's thesis, School of Computer Science, Mcgill University, Montreal (2015)
18. Thielecke, H.: Comparing control constructs by double-barrelled CPS. High.-Order Symb. Comput. **15** (2002)
19. Wadler, P.: The expression problem. The Java Genericity mailing list (1998)